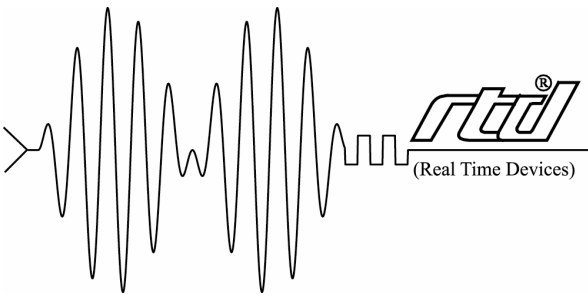
# DM5814/DM6814 Driver for Linux

# Driver Version 2.0.x User's Manual




RTD Embedded Technologies, Inc.
*"Accessing the Analog World"®*

SWM-640010021
rev B

ISO9001 and AS9100 Certified

**RTD Embedded Technologies, Inc.**
103 Innovation Boulevard
State College, PA 16803-0906

Phone: +1-814-234-8087
FAX: +1-814-234-5218

E-mail
sales@rtd.com
techsupport@rtd.com

web site
http://www.rtd.com

Revision History

| | |
|---|---|
| 03/01/2005 | Revision A issued<br>Documented for ISO9000 |
| 09/29/2005 | Revision B issued<br>Deleted discussion in “Interrupt Performance” section, replacing it with reference to application note SWM-640000021 |

DM5814/DM6814 Driver for Linux
Published by:

RTD Embedded Technologies, Inc.
103 Innovation Boulevard
State College, PA 16803-0906

Copyright 2005 by RTD Embedded Technologies, Inc.
All rights reserved
Printed in U.S.A.

The RTD logo and dataModule are registered trademarks of RTD Embedded Technologies. Linux is a registered trademark of Linus Torvalds. All other trademarks appearing in this document are the property of their respective owners.

# Table of Contents

**TABLE OF CONTENTS** ..... **4**

**INTRODUCTION** ..... **5**

**NOTATIONAL CONVENTIONS** ..... **6**

**INSTALLATION INSTRUCTIONS** ..... **7**

EXTRACTING THE SOFTWARE ..... 7
CONTENTS OF INSTALLATION DIRECTORY ..... 7
BUILDING THE DRIVER ..... 7
BUILDING THE LIBRARY ..... 8
BUILDING THE EXAMPLE PROGRAMS ..... 9

**INTERRUPT PERFORMANCE** ..... **10**

PERFORMANCE ISSUES ..... 10
REWRITING THE INTERRUPT HANDLER ..... 10

**USING THE API FUNCTIONS** ..... **12**

**FUNCTION REFERENCE** ..... **13**

**API FUNCTION GROUPS** ..... **14**

DIGITAL I/O ..... 14
INCREMENTAL ENCODER ..... 14
GENERAL ..... 14
INTERRUPT CONTROL AND STATUS ..... 14
TIMER/COUNTER CONTROL AND STATUS ..... 14

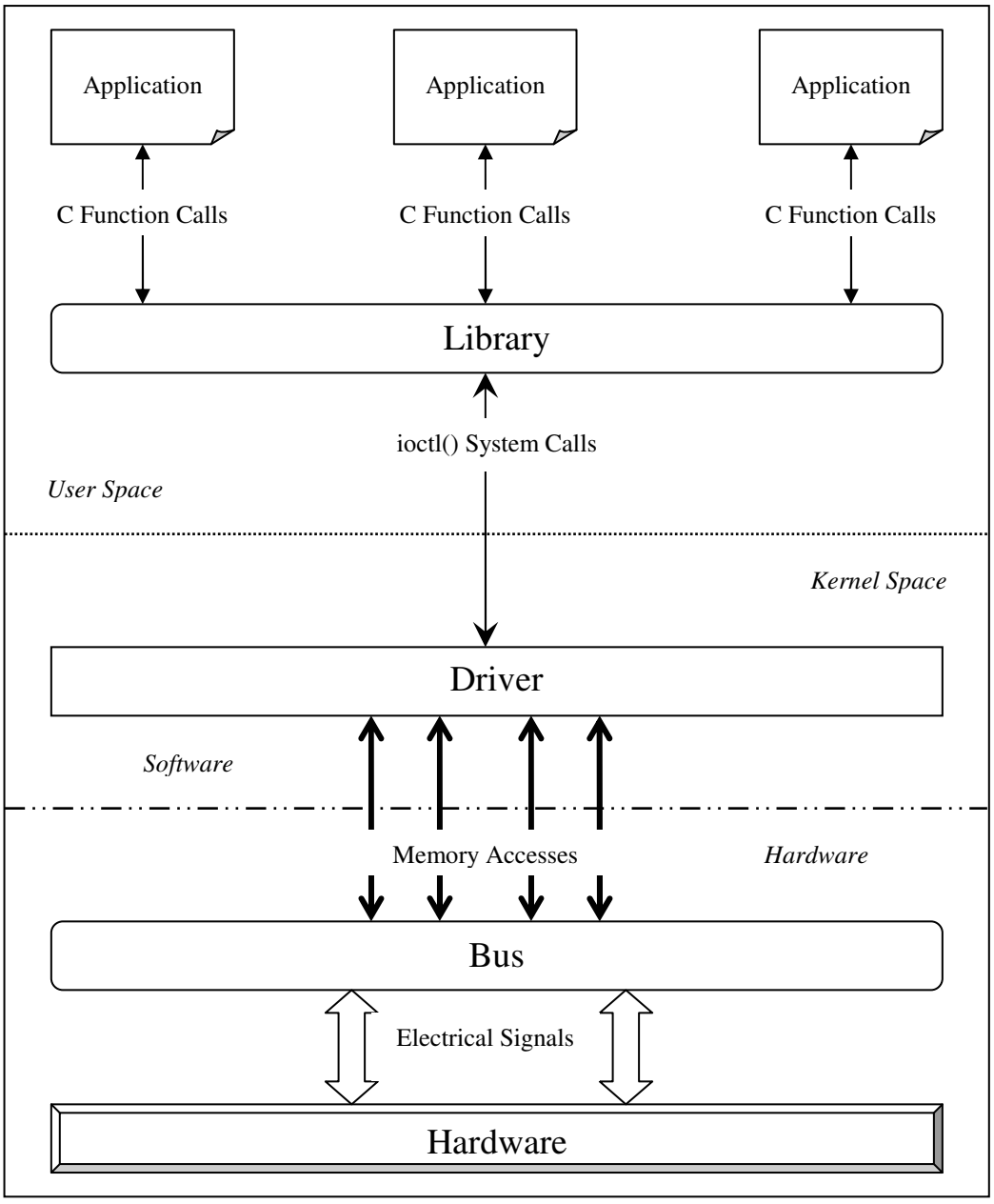
**ALPHABETICAL FUNCTION LISTING** ..... **15**

**LIMITED WARRANTY** ..... **36**

# Introduction

This document targets anyone wishing to write Linux applications for an RTD DM5814 or DM6814 dataModule. It provides information on building the software and about the Application Programming Interface used to communicate with the hardware and driver. Each high-level library function is described as well as any low-level ioctl() system call interface it may make use of.

The diagram below 1) provides a general overview of what hardware and software entities are involved in device access, 2) shows which units communicate with each other, and 3) illustrates the methods used to transfer data and control information.



## Notational Conventions

RTD Linux drivers are assigned version numbers. These version numbers take the form "**A**.**B**.**C**" where:

* **A** is the major release number. This will be incremented whenever major changes are made to the software. Changing the major release number requires updating the software manual.
* **B** is the minor release number. This will be incremented whenever minor, yet significant, changes are made to the software. Changing the minor release number requires updating the software manual.
* **C** is the patch level number. This will be incremented whenever very minor changes are made to the software. Changing the patch level number does not require updating the software manual.

In this document, you will see driver version numbers with a letter in them. For example, 2.0.x indicates that the topic being discussed is applicable to driver versions with a major release number of 2, a minor release number of 0, and any patch level number.

Occasionally you will notice text placed within the < and > characters, for example <installation path>. This indicates that the text represents something which depends upon choices you have made or upon your specific system configuration.

# Installation Instructions

## *Extracting the Software*

All software comes packaged in a gzip'd tar file named dm6814_Linux_v2.0.x.tar.gz. First, decide where you would like to place the software. Next, change your current directory to the directory in which you have chosen to install the software by issuing the command "cd <installation path>". Then, extract the software by issuing the "tar -xvzf <path to tar file>/dm6814_Linux_v2.0.x.tar.gz" command; this will create a directory dm6814_Linux_v2.0.x/ that contains all files comprising the software package.

## *Contents of Installation Directory*

Once the tar file is extracted, you should see the following files and directories within dm6814_Linux_v2.0.x/:

driver/
examples/
include/
lib/
CHANGES.TXT
LICENSE.TXT
README.TXT

The file CHANGES.TXT describes the changes made to the software for this release, as well as for previous releases. The file LICENSE.TXT provides details about the RTD end user license agreement which must be agreed to and accepted before using this software. The file README.TXT contains a general overview of the software and contact information should you experience problems, have questions, or need information. The directory driver/ contains the source code and Makefile for the driver. The directory examples/ holds the source code and Makefile for the example programs. The directory include/ contains all header files used by the driver, example programs, library, and your application programs. Library source code and Makefile reside in the directory lib/.

## *Building the Driver*

Driver source code uses files located in the kernel source tree. Therefore, you must have the full kernel source tree available in order to build the driver. The kernel source tree consumes a lot of disk space, on the order of 100 to 200 megabytes. Because production systems rarely contain this much disk space, you will probably use a development machine to compile the driver source code. The development system, which provides a full compilation environment, must be running the exact same version of the kernel as your production machine(s); otherwise the kernel module may not load or may load improperly. After the code is built, you can then move the resulting object files, libraries, and executables to the production system(s).

Building the driver consists of several steps: 1) compiling the source code, 2) loading the resulting kernel module into the kernel, and 3) creating hardware device files in the /dev directory. To perform any of the above steps, you must change your current directory to driver/. The file Makefile contains rules to assist you.

To compile the source code, issue the command “make”. The GNU C compiler gcc is used to build the driver code. This will create the driver object file, which is named rtd-dm6814.o on 2.4 kernels and rtd-dm6814.ko on 2.6 kernels.

Before the driver can be used, it must be loaded into the currently running kernel. Using the command “make insmod” will load the DM6814 driver into the kernel. This target assumes that:

* A single DM6814 is installed.
* The board's base I/O address is set to the factory default of 0x300.
* The DM6814 is jumpered to use IRQ 5.

If the previous assumptions do not match your hardware setup, you will need to edit the Makefile and change this rule to reflect your board configuration or manually issue an appropriate insmod command.

For the 2.4 kernel, when you load the kernel driver the message

"Warning: loading rtd-dm6814.o will taint the kernel: non-GPL license - Proprietary"

will be printed on your screen. You can safely ignore this message since it pertains to GNU General Public License (GPL) licensing issues rather than to driver operation.

For the 2.6 kernel, when you load the kernel driver, no warnings will appear on your screen. However, the warning

"module license 'Proprietary' taints kernel."

will be written to the system log when the module is loaded. You can safely ignore this message since it pertains to GNU General Public License (GPL) licensing issues rather than to driver operation.

The final step is to create /dev entries for the hardware. Versions of the driver prior to 2.0.0 always assumed a character device major number of 251 when registering the boards and creating the /dev entries. Instead, the driver now asks the kernel to dynamically assign a major number. Since this major number may change each time you load the driver, the best way to create the device files is to use the command "make devices"; this generates four files in /dev named rtd-dm6814-0 through rtd-dm6814-3.

Be aware that driver/Makefile uses the uname -r command to determine which kernel version it is running on. It does this to set up separate make rules and variables for the 2.4 and 2.6 kernels, which allows one set of targets to work on both kernels. Compiling the driver on a development machine which does not run the same kernel version as the production machine that will host your application almost certainly invites trouble.

If you ever need to unload the driver from the kernel, you can use the command "make rmmod".

## *Building the Library*

The example programs and your application use the DM6814 library, so it must be built before any of these can be compiled. To build the library, change your current directory to lib/ and issue the command “make”. The GNU C++ compiler g++ is used to compile the library source code. To prevent compatibility problems, any source code which makes use of library functions should also be built with g++.

The DM6814 library is statically linked and is created in the file librtd-dm6814.a.

## *Building the Example Programs*

The example programs may be compiled by changing your current directory to examples/ and issuing the command “make”, which builds all the example programs. If you wish to compile a subset of example programs, there are targets in Makefile to do so. For example, the command “make encoder-int external-int timers” will compile and link the source files encoder-int.cpp, external-int.cpp, and timers.cpp. The GNU C++ compiler g++ is used to compile the example program code.

# Interrupt Performance

## *Performance Issues*

Many factors exist outside of the driver software that may impact interrupt performance and throughput. For a discussion of these issues, please see the Application Note SWM-640000021 (Linux Interrupt Performance) on our web site.

## *Rewriting the Interrupt Handler*

Suppose that you have the following requirements: 1) the only interrupt being used is the P14 timer interrupt, 2) whenever a timer interrupt occurs, a one should be written to digital I/O port 0 bit 0, and 3) 1 microsecond after port 0 bit 0 is taken high, it should be taken low. Furthermore, assume the interrupts occur fast enough that dealing with them in user space is unreliable.

Because the interrupt handler provided with the driver offers generic services suitable for a wide range of purposes, it does not provide the functionality indicated above. Therefore to meet your requirements, you must rewrite the interrupt handler. A detailed description of writing interrupt handlers lies beyond the scope of this document. What follows is pseudo code for an interrupt handler which implements the desired behavior.

```
interrupt_handler() {

    /*
     * Determine interrupt status
     */

    read IRQ Status Register at offset 0x11;
    if bit 3 is not set in register
    then

        /*
         * Spurious interrupt
         */

        exit;
    end if

    /*
     * Acknowledge the interrupt by reading the Clear IRQ/IRQ Enable Register
     */

    read Clear IRQ/IRQ Enable Register at offset 0x10;

    /*
     * Write a one to port 0 bit 0.  This consists of two steps: 1) writing the
     * appropriate value to the Incremental Encoder 1 Chip Mode Register to select
     * digital I/O mode, and 2) writing to the Incremental Encoder 1 Digital I/O
     * Register.
     */

    write to Incremental Encoder 1 Chip Mode Register at offset 0x03;
    write 0x01 to Incremental Encoder 1 Digital I/O Register at offset 0x02;

    /*
     * Delay for one microsecond
     */

    udelay(1);
```

```
    /*
     * Take port 0 bit 0 low.  Note that Incremental Encoder 1 Chip Mode Register
     * has already been programmed above and does not need to be rewritten.
     */

    write 0x00 to Incremental Encoder 1 Digital I/O Register at offset 0x02;
}
```

## Using the API Functions

DM6814 hardware and the associated driver functionality can be accessed through the library API (Application Programming Interface) functions. Applications wishing to use library functions must include the include/dm6814_library.h header file and be statically linked with the lib/librtd-dm6814.a library file.

Because of changes made in driver version 2.0.0, existing source code which uses the library will not compile. Some of the areas requiring attention on your part are:

* All header files have been renamed. Users upgrading from a previous driver version will need to modify source code to include the appropriate header files.
* All header files have been relocated to include/. Be sure to update any files which contain hardcoded header file paths.
* 6814 has been appended to all library function names with the exception of class constructors and destructors.
* The driver and library no longer support interrupt notification via signals. The new notification paradigm provides functions which block in the kernel until an interrupt occurs.
* The driver now acknowledges all interrupts in the interrupt handler. Therefore, applications can no longer obtain direct board status. However, the driver caches the IRQ Status Register value and a program may obtain this instead.
* All classes have been coalesced into a single class DM6814Device.
* The DM6814 /dev entry file names have changed. For example, the device file previously named /dev/rtd/dm6814hr/device0 is now called /dev/rtd-dm6814-0.
* Obsolete functions have been deleted. You will need to identify these functions and replace them with alternate functionality if appropriate.
* Some functions require additional parameters to pass non-status information back to the caller, thus allowing the function return value to be an error indication.

The following function reference provides for each library routine a prototype, description, explanation of parameters, and return value or error code. By looking at a function's entry, you should gain an idea of: 1) why it would be used, 2) what it does, 3) what information is passed into it, 4) what information it passes back, 5) how to interpret error conditions that may arise, and 6) the ioctl() system call interface if the function makes use of a single ioctl() call.

Note that errno codes other than the ones indicated in the following pages may be set by the library functions. Please see the ioctl(2) man page for more information.

# Function Reference

## API Function Groups

### *Digital I/O*

ReadEncoderDIO6814
SetEncoderDIODirection6814
WriteEncoderDIO6814

### *Incremental Encoder*

ClearEncoderChip6814
EnableEncoder6814
EnableEncoderClear6814
EnableEncoderIrq6814
LoadEncoder6814
ReadEncoder6814

### *General*

CloseBoard6814
DM6814Device
~DM6814Device
GetDriverVersion6814
InitBoard6814
OpenBoard6814
ReadByte6814
WriteByte6814

### *Interrupt Control and Status*

EnableGetIntStatusWait6814
GetIntStatus6814
LoadIRQRegister6814
WaitForInterrupt6814

### *Timer/Counter Control and Status*

ClockDivisor6814
ClockMode6814
ReadTimerCounter6814
SetUserClock6814

# Alphabetical Function Listing

## ClearEncoderChip6814

bool ClearEncoderChip6814(uint8_t Encoder);

**Description:**

Clear the incremental encoder chip for the given incremental encoder. According to the hardware manual, clearing an encoder chip will 1) clear the encoder's IRQ status flag, 2) set digital I/O bits 0 and 1 to input, and 3) clear the encoder counter value.

**Parameters**:

Encoder: Incremental encoder chip to clear. Valid values are:
- 1 Incremental encoder 1
- 2 Incremental encoder 2
- 3 Incremental encoder 3

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the descriptions of the internal functions outb() and select_encoder_control_register() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

## ClockDivisor6814

bool ClockDivisor6814(uint8_t Timer, uint16_t Divisor);

**Description:**

Set the divisor for the given 8254 timer/counter.

*NOTE:* Before calling this function, you must ensure that the indicated timer/counter is set to be programmed least significant byte first then most significant byte.

**Parameters**:

Timer: The timer to operate on. Valid values are:
- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

Divisor: Counter divisor. Valid values are 0 through 65535.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Timer is not valid.

Please see the description of the internal function outb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

ClockMode6814

---

bool ClockMode6814(u_int8_t Timer, u_int8_t Mode);

**Description:**

Set the mode for the given 8254 timer/counter.

*NOTE:* This function puts the indicated timer/counter into binary mode.

*NOTE:* This function sets the timer/counter to read/load least significant byte first then most significant byte.

**Parameters**:

Timer: The timer to operate on. Valid values are:
- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

Mode: The counter mode to set. Valid values are:
- 0 Event count
- 1 Programmable one shot
- 2 Rate generator
- 3 Square wave rate generator
- 4 Software triggered strobe
- 5 Hardware triggered strobe

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Timer is not valid.

EINVAL Mode is not valid.

Please see the description of the internal function outb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

```
dm6814_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the 8254 Timer/Counter Control Word Register at base I/O address + 15
 */

ioctl_request.access_8.offset = 0x0F;

/*
 * Set timer to read/load least significant byte first then most significant
 * byte.  Because bit 0 is set to zero, this puts the timer in binary mode.
 */

ioctl_request.access_8.data = 0x30;

/*
 * Operate on timer/counter 1
 */

ioctl_request.access_8.offset |= 0x40;

/*
 * Put timer in rate generator mode
 */

ioctl_request.access_8.offset |= 0x04;

status = ioctl(file_descriptor, DM6814_IOCTL_WRITE_8_BITS, &ioctl_request);
```

## CloseBoard6814

bool CloseBoard6814(void);

**Description:**

Close a DM6814 device file.

*NOTE:* The kernel, when processing this function, performs all actions indicated in InitBoard6814().

**Parameters**:

None.

**Return Value:**

| | |
|---|---|
| true: | Success. |
| false: | Failure. Please see the close(2) man page for information on possible values errno may have in this case. |

**IOCTL Interface:**

None.

---

DM6814Device

---

DM6814Device(void);

**Description:**

DM6814Device class constructor.

**Parameters**:

None.

**Return Value:**

None. Constructors do not return a value.

**IOCTL Interface:**

None.

---

~DM6814Device

---

~DM6814Device(void);

**Description:**

DM6814Device class destructor.

*NOTE:* This function closes the DM6814 device file associated with the object.

**Parameters**:

None.

**Return Value:**

None. Destructors do not return a value.

**IOCTL Interface:**

None.

---

EnableEncoder6814

---

bool EnableEncoder6814(uint8_t Encoder, bool Enable);

**Description:**

Enable or disable the given incremental encoder.

**Parameters**:

Encoder: Incremental encoder to modify state of. Valid values are:
- 1 Incremental encoder 1
- 2 Incremental encoder 2
- 3 Incremental encoder 3

Enable: Flag indicating whether or not the encoder should be enabled. A value of false means disable the encoder. A value of true means enable the encoder.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

EnableEncoderClear6814

---

bool EnableEncoderClear6814(uint8_t Encoder, bool Enable);

**Description:**

Enable or disable encoder clear via bit 2 in the even-numbered digital I/O port for the given incremental encoder.

**Parameters**:

Encoder: Incremental encoder to modify clear state of. Valid values are:
1 Incremental encoder 1
2 Incremental encoder 2
3 Incremental encoder 3

Enable: Flag indicating whether or not encoder clear should be enabled. A value of false means disable encoder clear. A value of true means enable encoder clear.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

EnableEncoderIrq6814

---

bool EnableEncoderIrq6814(uint8_t Encoder, bool Enable);

**Description:**

Enable or disable incremental encoder interrupts for the given incremental encoder.

**Parameters**:

Encoder: Incremental encoder to modify interrupt state of. Valid values are:
1 Incremental encoder 1
2 Incremental encoder 2
3 Incremental encoder 3

Enable: Flag indicating whether or not interrupts should be enabled. A value of false means disable the encoder interrupt. A value of true means enable the encoder interrupt.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

EnableGetIntStatusWait6814

---

bool EnableGetIntStatusWait6814(bool Enable);

**Description:**

Enable or disable blocking I/O on a device file. This controls whether or not GetIntStatus6814() can block inside the kernel.

*NOTE:* The default behavior when a device file is opened is to disable blocking I/O.

**Parameters**:

Enable: Flag which controls whether or not blocking I/O is enabled. A value of false means blocking I/O is disabled and GetIntStatus6814() will never block in the kernel waiting for an interrupt to occur. A value of true means blocking I/O is enabled and GetIntStatus6814() can block in the kernel waiting for an interrupt to occur.

**Return Value:**

true: Success.

false: Failure. Please see the fcntl(2) man page for information on possible values errno may have in this case.

**IOCTL Interface:**

None.

---

GetDriverVersion6814

---

bool GetDriverVersion6814(uint32_t *version_p);

**Description:**

Get the driver version number. The version number is an unsigned integer encoding the major, minor, and patch level numbers.

*NOTE:* The driver version is encoded according to the formula

```
Version = (
    (MajorVersion << 16)
    |
    (MinorVersion << 8)
    |
    PatchLevelNumber
)
```

**Parameters**:

version_p: Address where version number should be stored. The contents of this memory is undefined if the function fails.

**Return Value:**

true: Success.

false: Failure. Please see the ioctl(2) man page for information on possible values errno may have in this case.

**IOCTL Interface:**

```
dm6814_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

status = ioctl(file_descriptor, DM6814_IOCTL_GET_DRIVER_VERSION, &ioctl_request);
if (status == 0) {
    u_int32_t  version;

    version = ioctl_request.version.driver_version;

    fprintf(stdout, "Major version: %d\n", ((version >> 16) & 0xF));
    fprintf(stdout, "Minor version: %d\n", ((version >> 8) & 0xF));
    fprintf(stdout, "Patch level: %d\n", (version & 0xF));
}
```

## GetIntStatus6814

bool GetIntStatus6814(uint32_t *int_count_p, uint8_t *status_reg_p);

**Description:**

Atomically obtain the driver's current interrupt count and cached IRQ Status Register value.

*NOTE:* This function clears the driver's cached IRQ Status Register value. The value will remain cleared until the next interrupt occurs.

*NOTE:* Only bits 0 through 3 in the cached IRQ Status Register value are returned.

*NOTE:* The macros P14_INT_OCCURRED(), ENC3_INT_OCCURRED(), ENC2_INT_OCCURRED(), and ENC1_INT_OCCURRED() should be used to examine specific Status Register value bits to determine the type(s) of interrupt(s) which occurred. Each macro returns true if the associated interrupt occurred and false if it did not.

*NOTE:* This function disables interrupts on the associated DM6814 device for a very brief time to obtain accurate status information. If you call the function repeatedly in a loop (such as when busy-waiting for an interrupt to occur), this can interfere with DM6814 interrupts. It is strongly suggested that you do not busy-wait for interrupts.

*NOTE:* If the device file has been set to use blocking I/O, this function will block inside the kernel until an interrupt occurs.

*NOTE:* If this function is being used to wait for interrupts, signals can wake up the process before an interrupt occurs. If a signal is delivered to the process during a wait, the application is responsible for dealing with the premature awakening in a reasonable manner.

*NOTE:* When this function is being used to wait for interrupts, it can be woken up by a signal before an interrupt occurs and an interrupt may be missed if signals are delivered rapidly enough or at inopportune times. To decrease the chances of this, it is strongly suggested that you 1) do not use signals or 2) minimize their use in your application.

*NOTE:* This function uses the read(2) system call to wait for an interrupt. read(2) can watch only a single file descriptor for activity.

**Parameters**:

int_count_p: Address where interrupt count should be stored. The contents of this memory is undefined if the function fails.

status_reg_p: Address where Status Register value should be stored. The contents of this memory is undefined if the function fails.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EBADMSG The expected number of bytes were not returned by read(2). This indicates a serious problem with the driver.

EINTR The process received a signal before an interrupt occurred. This occurs only if blocking I/O was set for the device file. This is not a fatal error but rather means the read should be retried.

EIO No IRQ line was allocated to the device when the driver was loaded.

Please see the read(2) man page for information on other possible values errno may have in this case.

**IOCTL Interface:**

None.

---

InitBoard6814

---

bool InitBoard6814(void);

**Description:**

Initialize a DM6814 device by performing the following actions: 1) disable IRQ sharing, 2) set positive edge P14 interrupt polarity, 3) disable P14 interrupts, 4) clear each of the incremental encoder chips, and 5) disable interrupts on each encoder chip.

**Parameters**:

None.

**Return Value:**

| | |
|---|---|
| true: | Success. |
| false: | Failure. Please see the descriptions of LoadIRQRegister6814(), ClearEncoderChip6814(), and EnableEncoderIrq6814() for information on possible values errno may have in this case. |

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

LoadEncoder6814

---

bool LoadEncoder6814(uint8_t Encoder, uint16_t Value);

**Description:**

Load a value into the given incremental encoder.

*NOTE:* Before loading a value into an incremental encoder, the encoder must be disabled.

**Parameters**:

| | |
|---|---|
| Encoder: | The incremental encoder to load. Valid values are:<br>1 Incremental encoder 1<br>2 Incremental encoder 2<br>3 Incremental encoder 3 |
| Value: | The value to load into the encoder. |

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the ioctl(2) man page for information on other possible values errno may have in this case.

**IOCTL Interface:**

```
dm6814_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the Incremental Encoder 1 LSB Register at base I/O address + 0
 */

ioctl_request.access_16.offset = 0x00;

/*
 * Load the encoder with the value 32768
 */

ioctl_request.access_16.data = 0x8000;

status = ioctl(file_descriptor, DM6814_IOCTL_WRITE_16_BITS, &ioctl_request);
```

---

LoadIRQRegister6814

---

bool LoadIRQRegister6814(uint8_t Value);

**Description:**

Load an 8-bit value into a board's Clear IRQ/IRQ Enable Register at base I/O address + 16.

*NOTE:* The DM6814 interrupt handler is not designed to process interrupts shared between devices. To avoid unpredictable behavior or worse, do not share an interrupt between devices.

*NOTE:* Interrupts do not need to be shared in order to use several interrupt sources on a single board. For example if you wish to use both P14 and encoder 1 interrupts on a single DM6814 device, then interrupts do not need to be shared.

*NOTE:* The macros ENABLE_IRQ_SHARING(), DISABLE_IRQ_SHARING(), POSITIVE_IRQ_POLARITY(), NEGATIVE_IRQ_POLARITY(), ENABLE_P14_IRQ(), and DISABLE_P14_IRQ() should be used to set or clear bits in the value passed to this function.

**Parameters**:

Value: Value to store in register. Valid values are 0 through 7.

**Return Value:**

true: Success.

false: Failure. Please see the description of the internal function outb() for information on possible values errno may have in this case.

**IOCTL Interface:**

```
dm6814_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the IRQ Enable Register at base I/O address + 16
 */

ioctl_request.access_8.offset = 0x10;

/*
 * Set up P14 interrupt
 */

ioctl_request.access_8.data = 0x00;
DISABLE_IRQ_SHARING(ioctl_request.access_8.data);
NEGATIVE_IRQ_POLARITY(ioctl_request.access_8.data);
ENABLE_P14_IRQ(ioctl_request.access_8.data);

status = ioctl(file_descriptor, DM6814_IOCTL_WRITE_8_BITS, &ioctl_request);
```

## OpenBoard6814

bool OpenBoard6814(uint32_t nDevice);

**Description:**

Open a DM6814 device file.

*NOTE:* Once a device file is open, it cannot be opened again until it is closed.

*NOTE:* The device file is opened with non-blocking I/O enabled.

*NOTE:* The kernel, when processing this function, performs all actions indicated in InitBoard6814().

**Parameters**:

nDevice: Minor number of DM6814 device file.

**Return Value:**

true: Success.

false: Failure. Please see the open(2) man page for information on possible values errno may have in this case.

**IOCTL Interface:**

None.

---

ReadByte6814

---

bool ReadByte6814(uint8_t offset, uint8_t *data_p);

**Description:**

Read an 8-bit value from the given offset within a DM6814 board's I/O memory.

*NOTE:* It is strongly suggested that you use other library functions instead of directly accessing a board's registers.

**Parameters**:

offset: Offset within I/O memory to read.

data_p: Address where data read should be stored. The contents of this memory is undefined if the function fails.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL offset is not valid.

EOPNOTSUPP offset is valid but it represents a write-only register.

Please see the ioctl(2) man page for information on other possible values errno may have in this case.

**IOCTL Interface:**

```
dm6814_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Read the IRQ Status Register at base I/O address + 17
 */
```

```
ioctl_request.access_8.offset = 0x11;

/*
 * This value does not matter because it is ignored making the request.
 * However after ioctl() returns, the structure member will contain the
 * register contents.
 */

ioctl_request.access_8.data = 0;

status = ioctl(file_descriptor, DM6814_IOCTL_READ_8_BITS, &ioctl_request);
if (status == 0) {
    fprintf(stdout, "IRQ Status Register: 0x%2x\n", ioctl_request.access_8.data);
}
```

## ReadEncoder6814

bool ReadEncoder6814(uint8_t Encoder, uint16_t *value_p);

**Description:**

Read the value in the given incremental encoder.

**Parameters**:

Encoder: The incremental encoder to read. Valid values are:

1 Incremental encoder 1
2 Incremental encoder 2
3 Incremental encoder 3

value_p: Address where encoder value should be stored. The contents of this memory is undefined if the function fails.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the ioctl(2) man page and the descriptions of the internal functions select_encoder_control_register() and outb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

## ReadEncoderDIO6814

bool ReadEncoderDIO6814(uint8_t Encoder, uint8_t *value_p);

**Description:**

Read the digital I/O lines for the given incremental encoder.

*NOTE:* Bits 0 and 1 in the value read are undefined if those port bits were not previously set to input.

*NOTE:* The port bits associated with bits 2 through 7 in the value read are always set to input.

**Parameters**:

Encoder: The incremental encoder to read digital I/O lines of. Valid values are:
- 1 Incremental encoder 1
- 2 Incremental encoder 2
- 3 Incremental encoder 3

value_p: Address where digital I/O value should be stored. The contents of this memory is undefined if the function fails.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the descriptions of the internal functions inb() and select_encoder_control_register() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

## ReadTimerCounter6814

bool ReadTimerCounter6814(uint8_t Timer, uint16_t *count_p);

**Description:**

Read the count for the given timer/counter.

**Parameters**:

Timer: The timer to operate on. Valid values are:
- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

count_p: Address where timer count should be stored. The contents of this memory is undefined if the function fails.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Timer is not valid.

Please see the descriptions of the internal functions outb() and inb() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

SetEncoderDIODirection6814

---

bool SetEncoderDIODirection6814(uint8_t Encoder, bool Bit0Output, bool Bit1Output);

**Description:**

Set the digital I/O line bit 0 and 1 directions for the given incremental encoder.

**Parameters**:

Encoder: The incremental encoder to set digital I/O line direction of. Valid values are:
- 1 Incremental encoder 1
- 2 Incremental encoder 2
- 3 Incremental encoder 3

Bit0Output: Flag indicating whether or not bit 0 should be set to output. A value of false means set bit 0 to input. A value of true means set bit 0 to output.

Bit1Output: Flag indicating whether or not bit 1 should be set to output. A value of false means set bit 1 to input. A value of true means set bit 1 to output.

**Return Value:**

| | |
|---|---|
| true: | Success. |
| false: | Failure with errno set as follows: |
| EINVAL | Encoder is not valid. |
| | Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case. |

**IOCTL Interface:**

This function makes use of several ioctl() requests.

---

SetUserClock6814

---

bool SetUserClock6814(uint8_t Timer, float InputRate, float OutputRate, float *actual_rate_p);

**Description:**

Set the given timer/counter into rate generator mode and program its divisor value based upon the specified input and output rates.

**Parameters**:

| | |
|---|---|
| Timer: | The timer to program. Valid values are:<br>0 Timer/counter 0<br>1 Timer/counter 1<br>2 Timer/counter 2 |
| InputRate: | Input clock rate to timer/counter. |
| OutputRate: | Desired output rate. |
| actual_rate_p: | Address where actual programmed frequency should be stored. The contents of this memory is undefined if the function fails. |

**Return Value:**

| | |
|---|---|
| true: | Success. |
| false: | Failure. Please see the descriptions of ClockDivisor6814() and ClockMode6814() for information on possible values errno may have in this case. |

**IOCTL Interface:**

This function makes use of several ioctl() requests.

bool WaitForInterrupt6814(void);

**Description:**

Wait for an interrupt to occur on a device.

*NOTE:* Signals can wake up the process before an interrupt occurs. If a signal is delivered to the process during a wait, the application is responsible for dealing with the premature awakening in a reasonable manner.

*NOTE:* Because this function can be woken up by a signal before an interrupt occurs, an interrupt may be missed if signals are delivered rapidly enough or at inopportune times. To decrease the chances of this, it is strongly suggested that you 1) do not use signals or 2) minimize their use in your application.

*NOTE:* This function uses the select(2) system call to wait for an interrupt. If necessary, select(2) can watch multiple file descriptors for activity.

**Parameters**:

None.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINTR — The process received a signal before an interrupt occurred. This is not a fatal error but rather means the wait should be retried.

EIO — No IRQ line was allocated to the device when the driver was loaded.

ENODATA — No signal was delivered and select(2) woke up without indicating that an interrupt occurred. This indicates serious problems within the driver.

Please see the select(2) man page for information on other possible values errno may have in this case.

**IOCTL Interface:**

None.

bool WriteByte6814(uint8_t offset, uint8_t data);

**Description:**

Write an 8-bit value to the given offset within a DM6814 board's I/O memory.

*NOTE:* It is strongly suggested that you use other library functions instead of directly accessing a board's registers.

**Parameters**:

offset: Offset within I/O memory to write.

data: Data to write.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL offset is not valid.

EOPNOTSUPP offset is valid but it represents a read-only register.

Please see the ioctl(2) man page for information on other possible values errno may have in this case.

**IOCTL Interface:**

```
dm6814_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the Incremental Encoder 1 Chip Mode Register at base I/O address + 3
 */

ioctl_request.access_8.offset = 0x03;

/*
 * Set up mode register value.  Bits 0 and 1 are zero which selects clear mode
 * when register at base I/O address + 2 is accessed.  Bit 2 is zero which sets
 * port 0 bit 0 to input.  Bit 3 is zero which sets port 0 bit 1 to input.  Bit 4
 * is zero which disables the encoder interrupt.  Bits 5 and 6 are one which
 * enables the encoder input.  Bit 7 is zero which disables encoder clear.
 */

ioctl_request.access_8.data = 0x60;

status = ioctl(file_descriptor, DM6814_IOCTL_WRITE_8_BITS, &ioctl_request);
```

bool WriteEncoderDIO6814(uint8_t Encoder, uint8_t Value);

**Description:**

Write the digital I/O lines for the given incremental encoder.

*NOTE:* To write to digital I/O port bits 0 and 1, these bits must have been previously set to output.

*NOTE:* The port bits associated with bits 2 through 7 in the value to write are ignored since they are always set to input.

**Parameters**:

Encoder: The incremental encoder to write digital I/O lines of. Valid values are:
- 1 Incremental encoder 1
- 2 Incremental encoder 2
- 3 Incremental encoder 3

Value: The digital value to write.

**Return Value:**

true: Success.

false: Failure with errno set as follows:

EINVAL Encoder is not valid.

Please see the descriptions of the internal functions outb() and select_encoder_control_register() for information on other possible values errno may have in this case.

**IOCTL Interface:**

This function makes use of several ioctl() requests.

## Example Programs Reference

| Name | Remarks |
|---|---|
| **basic-test** | Tests the basic functionality of the driver and library. |
| **digital-io** | Demonstrates reading from and writing to the digital I/O ports. |
| **dio-test** | Tests the library functions related to digital I/O. |
| **encoder-int** | Demonstrates how to use incremental encoder interrupts. |
| **encoders** | Demonstrates reading the incremental encoders. |
| **external-int** | Demonstrates how to use P14 external interrupts. |
| **interrupt-test** | Tests the library interrupt-related functions. |
| **int-wait-via-read** | Demonstrates waiting for interrupt notification via blocking I/O and the read(2) system call. |
| **int-wait-via-select** | Demonstrates waiting for interrupt notification via the select(2) system call. |
| **timer-int** | Demonstrates how to use P14 timer interrupts. |
| **timers** | Demonstrates using the 8254 timer/counters in rate generator and event count modes. |
| **timer-test** | Tests the basic functionality of the 8254 timer/counter library functions. |

## Limited Warranty

RTD Embedded Technologies, Inc. warrants the hardware and software products it manufactures and produces to be free from defects in materials and workmanship for one year following the date of shipment from RTD Embedded Technologies, INC. This warranty is limited to the original purchaser of product and is not transferable.

During the one year warranty period, RTD Embedded Technologies will repair or replace, at its option, any defective products or parts at no additional charge, provided that the product is returned, shipping prepaid, to RTD Embedded Technologies. All replaced parts and products become the property of RTD Embedded Technologies. Before returning any product for repair, customers are required to contact the factory for an RMA number.

THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY PRODUCTS WHICH HAVE BEEN DAMAGED AS A RESULT OF ACCIDENT, MISUSE, ABUSE (such as: use of incorrect input voltages, improper or insufficient ventilation, failure to follow the operating instructions that are provided by RTD Embedded Technologies, "acts of God" or other contingencies beyond the control of RTD Embedded Technologies), OR AS A RESULT OF SERVICE OR MODIFICATION BY ANYONE OTHER THAN RTD Embedded Technologies. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES ARE EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND RTD Embedded Technologies EXPRESSLY DISCLAIMS ALL WARRANTIES NOT STATED HEREIN. ALL IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES FOR MECHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED TO THE DURATION OF THIS WARRANTY. IN THE EVENT THE PRODUCT IS NOT FREE FROM DEFECTS AS WARRANTED ABOVE, THE PURCHASER'S SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. UNDER NO CIRCUMSTANCES WILL RTD Embedded Technologies BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, AND SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

RTD Embedded Technologies, Inc.
103 Innovation Boulevard
State College PA 16803-0906
USA
Our website: www.rtd.com